Haris Ziko, 1330011
Peter Lorenz, 1114658

Softwareparadigmen SS 2015, Übungsblatt 1

Beispiel 1:

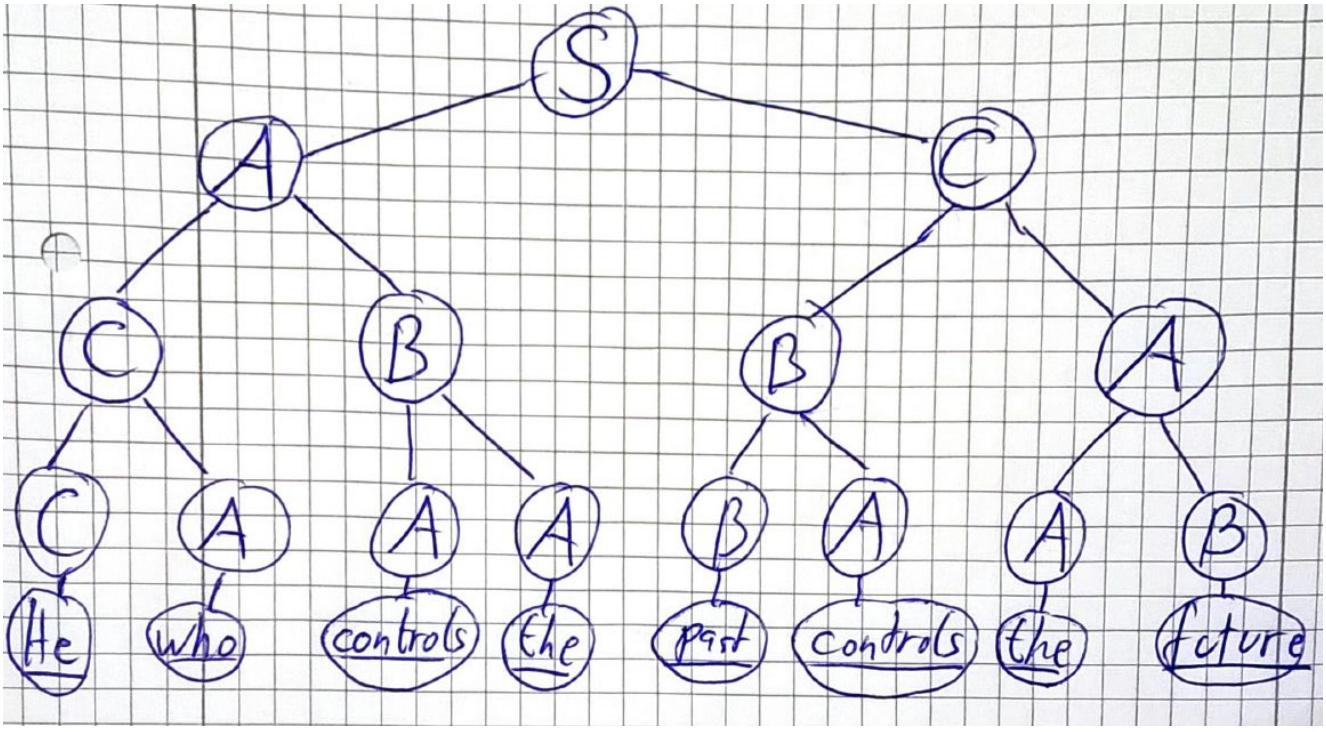
a. $L = \{a(cc|b)^n \ a^* | n > 0\}$

S → aB
B → ccC | bC
C → ccC | bC | A
A → aA | ε (terminiert)

b. $L = \{(bc)^* d^* a^{2n} \mid n \geq 0\}$

S→bcB | dD | aA
B→bcB | dD | aA
D→dD | aA
A→aA | ε

Beispiel 2:



Haris Ziko, 1330011
Peter Lorenz, 1114658

Beispiel 3:

a. **kontext-frei (Typ 2):**
$|\alpha| \leq |\beta|$ ist erfüllt.
Regel 2 weisst auf eine kontextfreie Grammatik hin, da ein Nonterminal von 2 Terminalen umschlossen ist. Man findet auch Regeln für reguläre Grammatiken. Diese sind jedoch in der Chomskyhierarchie eine Ebene tiefer und gehören deswegen zur kontextfreien Grammatik. Die Regeln der darunter liegenden Ebenen geerbt werden.

$$
\begin{aligned}
S &\rightarrow A\ \underline{char} \\
A &\rightarrow \underline{num}\ A\ \underline{num} \\
A &\rightarrow \underline{char}\ B \\
B &\rightarrow num \\
B &\rightarrow \epsilon
\end{aligned}
$$

b. **nicht möglich:**
a → b, terminal kann nicht zum Terminal uebergehen

$$
\begin{aligned}
S &\rightarrow \underline{a}\ A \\
A &\rightarrow \underline{a}\ A\ \underline{a} \\
A &\rightarrow \epsilon \\
\underline{a} &\rightarrow \underline{b} \\
B &\rightarrow \underline{b}\ B \\
B &\rightarrow \epsilon
\end{aligned}
$$

c. **kontext-frei (Typ 2):**
$|\alpha| \leq |\beta|$ ist erfüllt.
Auf der linken Seite '→' findet man nur NonTerminale. Beim Ersetzen der Non-Terminale ist egal, wo die Terminale stehen.

$$
\begin{aligned}
S &\rightarrow \underline{b}\ A \\
A &\rightarrow \underline{bb} A\ \underline{c} \\
A &\rightarrow B\underline{d} \\
B &\rightarrow \underline{d}\ B \\
B &\rightarrow \epsilon
\end{aligned}
$$

d. **kontext-sensitiv (Typ1):**
$|\alpha| \leq |\beta|$ ist erfüllt.
In der letzten Regel findet man auf der linken Seite des '→' einen Hinweis für kontext-sensitive Grammatiken, da diese von einen Non-Terminal und Terminal abhängig ist.

$$
\begin{aligned}
S &\rightarrow \underline{x}\ X\ Y \\
X &\rightarrow \underline{x}\ X\ Y \\
Y &\rightarrow Y\ \underline{y} \\
Y &\rightarrow \underline{y} \\
X\ \underline{y} &\rightarrow \underline{x}\ \underline{y}\ \underline{z}
\end{aligned}
$$

e. **unrestricted (Typ 0):**
$|\alpha| \leq |\beta|$ ist **nicht** erfüllt.
Regel 5 weißt eine unerlaubte Struktur auf.
$|\alpha| \leq |\beta|$ ist die Bedingung, jedoch die Regel 5 sieht folgendermaßen aus: Y<u>num</u> → <u>char</u>

$$
\begin{aligned}
S &\rightarrow \underline{char}\ X \\
X &\rightarrow \underline{num}\ X\ \underline{num} \\
X &\rightarrow Y\ \underline{num} \\
X\underline{num} &\rightarrow \underline{char}\ Y \\
Y\underline{num} &\rightarrow \underline{char} \\
Y &\rightarrow \underline{char} \\
Y &\rightarrow \epsilon
\end{aligned}
$$

f. **regulär (Typ 3):**
$|\alpha| \leq |\beta|$ ist erfüllt.
Die darunter angeführten Regeln sind alle auf den gleichen Prinzip aufgebaut: Variablen werden abgeleitet zu Terminalen verbunden mit Non-Terminalen.

$$
\begin{aligned}
S &\rightarrow \underline{a}\ M \\
M &\rightarrow \underline{m}\ M \\
M &\rightarrow \underline{m}\ N \\
N &\rightarrow \underline{nn}\ N \\
N &\rightarrow \epsilon
\end{aligned}
$$

Beispiel 4:

	FIRST	**FOLLOW**
S	{a,b,c,char, ε }	{$}
A	{a, ε }	{b,c,char,$}
B	{b,c,char, ε }	{$}
C	{num,char}	{$}
D	{char}	{char,num,$}

LL(1) Tabelle:

	a	**b**	**c**	**num**	**char**	**$**
S	S→AB	S→AB	S→AB		S→AB	S→AB
A	A→aA	A→ε	A→ε		A→ε	A→ε
B		B→bC	B→cC		B→DC	B→ε
C				C→numD	C→charC	
D					D→char	

Beispiel 5:

a. var one : String=101

Stack	**Input**	**Produktionsregel**
$ S	var one : String=101$	S →AB
$ BA	var one : String=101$	A → CN:
$ B:NC	var one : String=101$	C → var
$ B:N	one : String=101$	N → one
$ B:	: String=101$	✓
$ B	String=101$	B → String=”V”
$ ”V”=String	String=101$	✓
$ “V”=	=101$	✓
$ “V”	101$	**No Matching!**

b. val two : Int=11

Stack	Input	Produktionsregel
$ S	val two : Int=11$	S → AB
$ BA	val two : Int=11$	A → CN:
$ B:NC	val two : Int=11$	C → val
$ B:N	two : Int=11$	N → two
$ B:	: Int=11$	✓
$ B	Int=11$	B → Int=U
$ U=Int	Int=11$	✓
$ U=	=11$	✓
$ U	11$	U → 1V
$ V1	11$	✓
$ V	1$	V → 1V
$ V1	1$	✓
$ V	ε$	V → ε
$	$	**Matching!**

Beispiel 6: äquivalente Grammatik. Zeigen Sie, warum folgende Grammatik keine LL(1) ist:

Definition (laut Skriptum):

Definition 1.17 (LL(1)-Grammatik)**:** Eine kontextfreie Grammatik G ist eine LL(1)-Grammatik (ist in LL(1)-Form) wenn sie

- keine Linksrekursionen enthält (z.B. $L \to L\underline{a}$)
- keine Produktionen mit gleichen Präfixen für die selbe linke Seite enthält (z.B. $A \to \underline{a}B$ und $A \to \underline{a}C$)
- ermöglicht immer in einem Schritt (d.h. nur mit Kenntnis des nächsten Tokens) zu entscheiden, welche Produktionsregel zur Ableitung verwendet werden muss.

- **Indirekte** Linksrekursionen — A → aBb sowie B → c
- **Direkte** Linksrekursionen — A → Aa und A → b
- **Linksfaktorisierungen** — A → aB sowie A → aC

Aufgabe 6:

$S \rightarrow AB$
$A \rightarrow Cz$
$A \rightarrow \varepsilon$
$C \rightarrow xD$
$C \rightarrow yE$
$D \rightarrow y$
$E \rightarrow x$
$B \rightarrow +$
$B \rightarrow G+F$
$F \rightarrow :$
$F \rightarrow .$
$G \rightarrow H0$
$G \rightarrow 1$
$H \rightarrow GF$

Diese gegebene Grammatik ist keine LL(1) Grammatik. Gründe:

- Bei (4) und (5) scheint eine Linksfaktorisierung auf
- Bei (12) kommt eine Linksrekursion vor

Wir müssen sie in einer LL(1) Grammatik umformen.

So schauts dann aus:

(1) S → AB
(2) A → Cz
(3) A → ε
(4) C → xD
(5) D → y
(6) D → x
(7) B → +
(8) B → G + F
(9) F → :
(10) F → .
(11) G → 1H
(12) H → F0H
(13) H → ε

Jetzt haben wir eine LL(1) Grammatik vorliegen.

	+	.	0	1	:	x	y	z	$
S	S → AB			S → AB		S → AB			
A	A → ε			A → ε		A → Cz			
B	B → +			B → G+F					
C						C → xD			
D						D → x	D → y		
F		F → .			F → :				
G				G → 1H					
H	H → ε	H → F0H			H → F0H				

Beispiel 7: Scala

```scala
import scala.util.parsing.combinator.JavaTokenParsers

sealed trait Expression1a
case class String1a(str: String) extends Expression1a // wie in c++ das vererben

sealed trait Expression1b // 2+3i// -4-5*i// +4-i// i// 4
case class String1b(str: String) extends Expression1b

/**
 *  Parser definition using the Scala Parser Combinator Library
 *
class ExpParser1a extends JavaTokenParsers {
  def expression: Parser[Expression1a] = string1a // | mult | add

  private val string1a : Parser[String1a] =
    "a(cc|b)+a*".r ^^ {str => String1a(str.toString)}
}

object ParseExpression1a extends ExpParser1a {

  // The apply method is used to overload the () syntax
  // You can use it like this: ParseExpression(stringToParse)
  def apply(s: String): ParseResult[Expression1a] = {
    parseAll(expression, s)
  }
}

class ExpParser1b extends JavaTokenParsers {
  //An expression is either a number, a mult or a add
  // | the result is either the left or the right result, the left parser is tried first
  def expression: Parser[Expression1b] = string1b

  private val string1b : Parser[String1b] =
    (("((\\+|-)?([1-9][0-9]*|0|-[1-9][0-9]*))?" +
     "((\\+|-)?([1-9][0-9]*|0|-[1-9][0-9]*)?i)?").r ^^ {str => String1b(str.toString)}
}

object ParseExpression1b extends ExpParser1b {

  // The apply method is used to overload the () syntax
  // You can use it like this: ParseExpression(stringToParse)
  def apply(s: String): ParseResult[Expression1b] = {
    parseAll(expression, s)
  }
}

  /**
   *  Main object with a method to interpret an Expression and a main method to run the
program
```

```
   */
  object SWPbsp7 {
    def interpret1a(expression: Expression1a): String = expression match {
      case String1a(str) => str
    }

    def interpret1b(expression: Expression1b): String = expression match {
      case String1b(str) => str
    }

    def autotest7a : Unit =
    {
      var loop : Int = 0
      val teststrings : Array[String] = Array("accbaaa","accccbcc","abbbbccaaaaa")
      while (loop < teststrings.length)
      {
        println("Testing # "+loop+": "+teststrings(loop))
        val input = teststrings(loop)
        val expression = ParseExpression1a(input).get // get returns the Expression if the
                                      parsing was successful, otherwise an exception is thrown
      val resultValue = interpret1a(expression)
        println(resultValue)
        loop += 1
      }
    }

    def autotest7b : Unit =
    {
      var loop : Int = 0
      val teststrings : Array[String] =
Array("4+5i","-5+i","i","-i","5","-5","-5-i","4567-234234i","+i")
      while (loop < teststrings.length)
      {
        println("Testing # "+loop+": "+teststrings(loop))
        val input = teststrings(loop)
        val expression = ParseExpression1b(input).get
        val resultValue = interpret1b(expression)
        println(resultValue)
        loop += 1
      }
    }

    def main(args: Array[String]): Unit = {
      args(0) match
      {
        case "7a" =>
          args(1) match
          {
            case "auto" =>
              autotest7a
```

```
            case _ =>
              val input = args(1)
              val expression = ParseExpression1a(input).get
              val resultValue = interpret1a(expression)
              print(resultValue)
          }
        case "7b" =>
          args(1) match
          {
            case "auto" =>
              autotest7b
            case _ =>
              val input = args(1)
              val expression = ParseExpression1b(input).get
              val resultValue = interpret1b(expression)
              print(resultValue)
          }
      }
    }
  }
```